Tutorial: Run a parallel workload with Azure Batch using the .NET API

- Article
- 04/19/2023
- 17 contributors

In this article

1. Prerequisites
2. Sign in to Azure
3. Add an application package
4. Get account credentials

Use Azure Batch to run large-scale parallel and high-performance computing (HPC) batch jobs efficiently in Azure. This tutorial walks through a C# example of running a parallel workload using Batch. You learn a common Batch application workflow and how to interact programmatically with Batch and Storage resources.

- Add an application package to your Batch account.
- Authenticate with Batch and Storage accounts.
- Upload input files to Storage.
- Create a pool of compute nodes to run an application.
- Create a job and tasks to process input files.
- Monitor task execution.
- Retrieve output files.

In this tutorial, you convert MP4 media files to MP3 format, in parallel, by using the ffmpeg open-source tool.

If you don't have an Azure subscription, create an Azure free account before you begin.

Prerequisites

- Visual Studio 2017 or later, or .NET Core SDK for Linux, macOS, or Windows.
- A Batch account and a linked Azure Storage account. To create these accounts, see the Batch quickstart guides for the Azure portal or Azure CLI.

- Download the appropriate version of ffmpeg for your use case to your local computer. This tutorial and the related sample app use the [Windows 64-bit full-build version of ffmpeg 4.3.1](). For this tutorial, you only need the zip file. You do not need to unzip the file or install it locally.

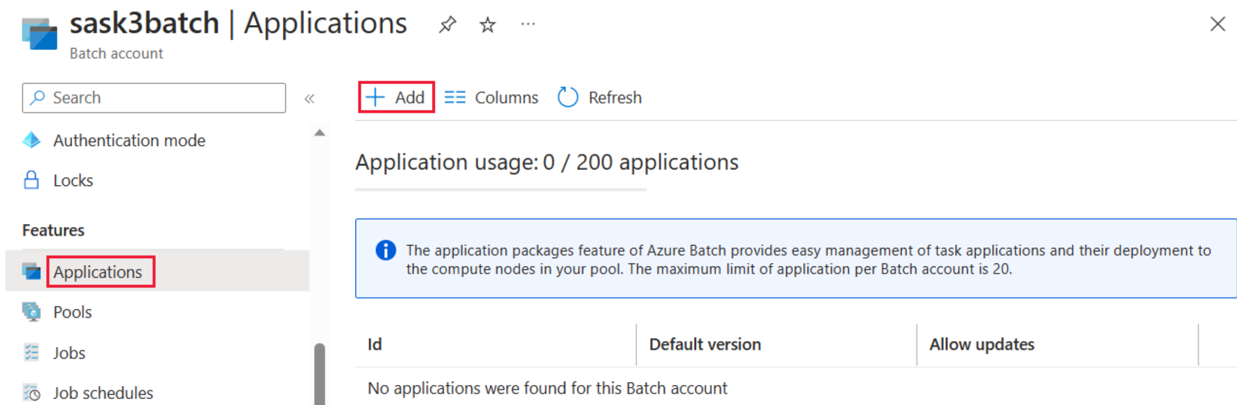
Sign in to Azure

Sign in to [the Azure portal]().

Add an application package

Use the Azure portal to add ffmpeg to your Batch account as an [application package](). Application packages help you manage task applications and their deployment to the compute nodes in your pool.

1. In the Azure portal, click **More services** > **Batch accounts**, and select the name of your Batch account.
2. Click **Applications** > **Add**.



3. Enter *ffmpeg* in the **Application Id** field, and a package version of *4.3.1* in the **Version** field. Select the ffmpeg zip file that you downloaded, and then select **Submit**. The ffmpeg application package is added to your Batch account.

New application ···

Application Id * ⓘ

ffmpeg

Version * ⓘ

4.3.1

Application package * ⓘ

Submit

Get account credentials

For this example, you need to provide credentials for your Batch and Storage accounts. A straightforward way to get the necessary credentials is in the Azure portal. (You can also get these credentials using the Azure APIs or command-line tools.)

1. Select **All services** > **Batch accounts**, and then select the name of your Batch account.
2. To see the Batch credentials, select **Keys**. Copy the values of **Batch account**, **URL**, and **Primary access key** to a text editor.
3. To see the Storage account name and keys, select **Storage account**. Copy the values of **Storage account name** and **Key1** to a text editor.

Download and run the sample app

Download the sample app

Download or clone the sample app from GitHub. To clone the sample app repo with a Git client, use the following command:

Copy

```
git clone https://github.com/Azure-Samples/batch-dotnet-ffmpeg-tutorial.git
```

Navigate to the directory that contains the Visual Studio solution file *BatchDotNetFfmpegTutorial.sln*.

Open the solution file in Visual Studio, and update the credential strings in *Program.cs* with the values you obtained for your accounts. For example:

C#Copy
```
// Batch account credentials
private const string BatchAccountName = "yourbatchaccount";
private const string BatchAccountKey  =
"xxxxxxxxxxxxxxxxE+yXrRvJAqT9BlXwwo1CwF+SwAYOxxxxxxxxxxxxxxxx43pXi/gdiATkvbpLRl3x14pc
EQ==";
private const string BatchAccountUrl  =
"https://yourbatchaccount.yourbatchregion.batch.azure.com";

// Storage account credentials
private const string StorageAccountName = "yourstorageaccount";
private const string StorageAccountKey  =
"xxxxxxxxxxxxxxxxy4/xxxxxxxxxxxxxxxxfwpbIC5aAWA8wDu+AFXZB827Mt9lybZB1nUcQbQiUrkPtilK5
BQ==";
```

Note

To simplify the example, the Batch and Storage account credentials appear in clear text. In practice, we recommend that you restrict access to the credentials and refer to them in your code using environment variables or a configuration file. For examples, see the Azure Batch **code samples repo**.

Also, make sure that the ffmpeg application package reference in the solution matches the identifier and version of the ffmpeg package that you uploaded to your Batch account. For example, `ffmpeg` and `4.3.1`.

C#Copy
```
const string appPackageId = "ffmpeg";
const string appPackageVersion = "4.3.1";
```

Build and run the sample project

Build and run the application in Visual Studio, or at the command line with the `dotnet build` and `dotnet run` commands. After running the application, review the code to learn what each part of the application does. For example, in Visual Studio:

1. Right-click the solution in Solution Explorer and select **Build Solution**.

2. Confirm the restoration of any NuGet packages, if you're prompted. If you need to download missing packages, ensure the [NuGet Package Manager]() is installed.
3. Run the solution. When you run the sample application, the console output is similar to the following. During execution, you experience a pause at `Monitoring all tasks for 'Completed' state, timeout in 00:30:00...` while the pool's compute nodes are started.

Copy

```
Sample start: 11/19/2018 3:20:21 PM

Container [input] created.
Container [output] created.
Uploading file LowPriVMs-1.mp4 to container [input]...
Uploading file LowPriVMs-2.mp4 to container [input]...
Uploading file LowPriVMs-3.mp4 to container [input]...
Uploading file LowPriVMs-4.mp4 to container [input]...
Uploading file LowPriVMs-5.mp4 to container [input]...
Creating pool [WinFFmpegPool]...
Creating job [WinFFmpegJob]...
Adding 5 tasks to job [WinFFmpegJob]...
Monitoring all tasks for 'Completed' state, timeout in 00:30:00...
Success! All tasks completed successfully within the specified timeout period.
Deleting container [input]...

Sample end: 11/19/2018 3:29:36 PM
Elapsed time: 00:09:14.3418742
```

Go to your Batch account in the Azure portal to monitor the pool, compute nodes, job, and tasks. For example, to see a heat map of the compute nodes in your pool, click **Pools** > **WinFFmpegPool**.

When tasks are running, the heat map is similar to the following:

Typical execution time is approximately *10 minutes* when you run the application in its default configuration. Pool creation takes the most time.

Retrieve output files

You can use the Azure portal to download the output MP3 files generated by the ffmpeg tasks.

1. Click **All services** > **Storage accounts**, and then click the name of your storage account.
2. Click **Blobs** > *output*.
3. Right-click one of the output MP3 files and then click **Download**. Follow the prompts in your browser to open or save the file.

Although not shown in this sample, you can also download the files programmatically from the compute nodes or from the storage container.

Review the code

The following sections break down the sample application into the steps that it performs to process a workload in the Batch service. Refer to the file *Program.cs* in the solution while you read the rest of this article, since not every line of code in the sample is discussed.

Authenticate Blob and Batch clients

To interact with the linked storage account, the app uses the Azure Storage Client Library for .NET. It creates a reference to the account with CloudStorageAccount, authenticating using shared key authentication. Then, it creates a CloudBlobClient.

C#Copy
```
// Construct the Storage account connection string
string storageConnectionString =
String.Format("DefaultEndpointsProtocol=https;AccountName={0};AccountKey={1}",
                                StorageAccountName, StorageAccountKey);

// Retrieve the storage account
CloudStorageAccount storageAccount =
CloudStorageAccount.Parse(storageConnectionString);

CloudBlobClient blobClient = storageAccount.CreateCloudBlobClient();
```

The app creates a BatchClient object to create and manage pools, jobs, and tasks in the Batch service. The Batch client in the sample uses shared key authentication. Batch also supports authentication through Azure Active Directory to authenticate individual users or an unattended application.

C#Copy
```
BatchSharedKeyCredentials sharedKeyCredentials = new
BatchSharedKeyCredentials(BatchAccountUrl, BatchAccountName, BatchAccountKey);

using (BatchClient batchClient = BatchClient.Open(sharedKeyCredentials))
...
```

Upload input files

The app passes the `blobClient` object to the `CreateContainerIfNotExistAsync` method to create a storage container for the input files (MP4 format) and a container for the task output.

C#Copy
```
CreateContainerIfNotExistAsync(blobClient, inputContainerName);
CreateContainerIfNotExistAsync(blobClient, outputContainerName);
```

Then, files are uploaded to the input container from the local *InputFiles* folder. The files in storage are defined as Batch ResourceFile objects that Batch can later download to compute nodes.

Two methods in *Program.cs* are involved in uploading the files:

- `UploadFilesToContainerAsync`: Returns a collection of `ResourceFile` objects and internally calls `UploadResourceFileToContainerAsync` to upload each file that is passed in the `inputFilePaths` parameter.
- `UploadResourceFileToContainerAsync`: Uploads each file as a blob to the input container. After uploading the file, it obtains a shared access signature (SAS) for the blob and returns a `ResourceFile` object to represent it.

C#Copy
```
string inputPath = Path.Combine(Environment.CurrentDirectory, "InputFiles");

List<string> inputFilePaths = new
List<string>(Directory.GetFileSystemEntries(inputPath, "*.mp4",
    SearchOption.TopDirectoryOnly));

List<ResourceFile> inputFiles = await UploadFilesToContainerAsync(
  blobClient,
  inputContainerName,
  inputFilePaths);
```

For details about uploading files as blobs to a storage account with .NET, see Upload, download, and list blobs using .NET.

Create a pool of compute nodes

Next, the sample creates a pool of compute nodes in the Batch account with a call to `CreatePoolIfNotExistAsync`. This defined method uses the BatchClient.PoolOperations.CreatePool method to set the number of nodes, VM size, and a pool configuration. Here, a VirtualMachineConfiguration object specifies an ImageReference to a Windows Server image published in the Azure Marketplace. Batch supports a wide range of VM images in the Azure Marketplace, as well as custom VM images.

The number of nodes and VM size are set using defined constants. Batch supports dedicated nodes and Spot nodes, and you can use either or both in your pools. Dedicated nodes are reserved for your pool. Spot nodes are offered at a reduced price from surplus VM capacity in Azure. Spot nodes become unavailable if Azure does not have enough capacity. The sample by default creates a pool containing only 5 Spot nodes in size *Standard_A1_v2*.

Note

Be sure you check your node quotas. See **Batch service quotas and limits** for instructions on how to create a quota request.

The ffmpeg application is deployed to the compute nodes by adding an ApplicationPackageReference to the pool configuration.

The CommitAsync method submits the pool to the Batch service.

C#Copy
```
ImageReference imageReference = new ImageReference(
    publisher: "MicrosoftWindowsServer",
    offer: "WindowsServer",
    sku: "2016-Datacenter-smalldisk",
    version: "latest");

VirtualMachineConfiguration virtualMachineConfiguration =
    new VirtualMachineConfiguration(
    imageReference: imageReference,
    nodeAgentSkuId: "batch.node.windows amd64");

pool = batchClient.PoolOperations.CreatePool(
    poolId: poolId,
    targetDedicatedComputeNodes: DedicatedNodeCount,
    targetLowPriorityComputeNodes: LowPriorityNodeCount,
    virtualMachineSize: PoolVMSize,
    virtualMachineConfiguration: virtualMachineConfiguration);

pool.ApplicationPackageReferences = new List<ApplicationPackageReference>
    {
    new ApplicationPackageReference {
    ApplicationId = appPackageId,
    Version = appPackageVersion}};

await pool.CommitAsync();
```

Create a job

A Batch job specifies a pool to run tasks on and optional settings such as a priority and schedule for the work. The sample creates a job with a call to `CreateJobAsync`. This defined method uses the BatchClient.JobOperations.CreateJob method to create a job on your pool.

The CommitAsync method submits the job to the Batch service. Initially the job has no tasks.

C#Copy
```
CloudJob job = batchClient.JobOperations.CreateJob();
job.Id = JobId;
job.PoolInformation = new PoolInformation { PoolId = PoolId };

await job.CommitAsync();
```

Create tasks

The sample creates tasks in the job with a call to the `AddTasksAsync` method, which creates a list of [CloudTask](#) objects. Each `CloudTask` runs ffmpeg to process an input `ResourceFile` object using a [CommandLine](#) property. ffmpeg was previously installed on each node when the pool was created. Here, the command line runs ffmpeg to convert each input MP4 (video) file to an MP3 (audio) file.

The sample creates an [OutputFile](#) object for the MP3 file after running the command line. Each task's output files (one, in this case) are uploaded to a container in the linked storage account, using the task's [OutputFiles](#) property. Previously in the code sample, a shared access signature URL (`outputContainerSasUrl`) was obtained to provide write access to the output container. Note the conditions set on the `outputFile` object. An output file from a task is only uploaded to the container after the task has successfully completed (`OutputFileUploadCondition.TaskSuccess`). See the full [code sample](#) on GitHub for further implementation details.

Then, the sample adds tasks to the job with the [AddTaskAsync](#) method, which queues them to run on the compute nodes.

Replace the executable's file path with the name of the version that you downloaded. This sample code uses the example `ffmpeg-4.3.1-2020-11-08-full_build`.

C#Copy

```
 // Create a collection to hold the tasks added to the job.
List<CloudTask> tasks = new List<CloudTask>();

for (int i = 0; i < inputFiles.Count; i++)
{
    string taskId = String.Format("Task{0}", i);

    // Define task command line to convert each input file.
    string appPath = String.Format("%AZ_BATCH_APP_PACKAGE_{0}#{1}%", appPackageId,
appPackageVersion);
    string inputMediaFile = inputFiles[i].FilePath;
    string outputMediaFile = String.Format("{0}{1}",
        System.IO.Path.GetFileNameWithoutExtension(inputMediaFile),
        ".mp3");
    string taskCommandLine = String.Format("cmd /c {0}\\ffmpeg-4.3.1-2020-09-21-
full_build\\bin\\ffmpeg.exe -i {1} {2}", appPath, inputMediaFile, outputMediaFile);

    // Create a cloud task (with the task ID and command line)
    CloudTask task = new CloudTask(taskId, taskCommandLine);
    task.ResourceFiles = new List<ResourceFile> { inputFiles[i] };

    // Task output file
    List<OutputFile> outputFileList = new List<OutputFile>();
```

```
    OutputFileBlobContainerDestination outputContainer = new
OutputFileBlobContainerDestination(outputContainerSasUrl);
    OutputFile outputFile = new OutputFile(outputMediaFile,
       new OutputFileDestination(outputContainer),
       new OutputFileUploadOptions(OutputFileUploadCondition.TaskSuccess));
    outputFileList.Add(outputFile);
    task.OutputFiles = outputFileList;
    tasks.Add(task);
}

// Add tasks as a collection
await batchClient.JobOperations.AddTaskAsync(jobId, tasks);
return tasks
```

Monitor tasks

When Batch adds tasks to a job, the service automatically queues and schedules them for execution on compute nodes in the associated pool. Based on the settings you specify, Batch handles all task queuing, scheduling, retrying, and other task administration duties.

There are many approaches to monitoring task execution. This sample defines a `MonitorTasks` method to report only on completion and task failure or success states. The `MonitorTasks` code specifies an ODATADetailLevel to efficiently select only minimal information about the tasks. Then, it creates a TaskStateMonitor, which provides helper utilities for monitoring task states. In `MonitorTasks`, the sample waits for all tasks to reach `TaskState.Completed` within a time limit. Then it terminates the job and reports on any tasks that completed but may have encountered a failure such as a non-zero exit code.

C#Copy

```
TaskStateMonitor taskStateMonitor = batchClient.Utilities.CreateTaskStateMonitor();
try
{
    await taskStateMonitor.WhenAll(addedTasks, TaskState.Completed, timeout);
}
catch (TimeoutException)
{
    batchClient.JobOperations.TerminateJob(jobId);
    Console.WriteLine(incompleteMessage);
    return false;
}
batchClient.JobOperations.TerminateJob(jobId);
 Console.WriteLine(completeMessage);
...
```

Clean up resources

After it runs the tasks, the app automatically deletes the input storage container it created, and gives you the option to delete the Batch pool and job. The BatchClient's JobOperations and PoolOperations classes both have corresponding delete methods, which are called if you confirm deletion. Although you're not charged for jobs and tasks themselves, you are charged for compute nodes. Thus, we recommend that you allocate pools only as needed. When you delete the pool, all task output on the nodes is deleted. However, the output files remain in the storage account.

When no longer needed, delete the resource group, Batch account, and storage account. To do so in the Azure portal, select the resource group for the Batch account and click **Delete resource group**.

Next steps

In this tutorial, you learned how to:

- Add an application package to your Batch account.
- Authenticate with Batch and Storage accounts.
- Upload input files to Storage.
- Create a pool of compute nodes to run an application.
- Create a job and tasks to process input files.
- Monitor task execution.
- Retrieve output files.

For more examples of using the .NET API to schedule and process Batch workloads, see the Batch C# samples on GitHub.